DESIGN AND IMPLEMENTATION OF DATA STREAM PROCESSING APPLICATIONS

Edwin Kwan, Janusz R. Getta
School of Computer Science and Software Engineering, University of Wollongong,Wollongong, Australia

Ehsan Vossough
Department of Computing and Mathematics, University of Western Sydney, Campbelltown, Australia

Keywords: Data stream, data stream management system, data stream application, processing data streams.

Abstract: Processing of data streams requires the continuous processing of end-user applications over the long and steadily increasing sequences of data items. This work considers the design and implementation of data stream processing applications in the domains where the limited computational resources, constraints imposed on the implementation techniques and specific properties of applications exclude the use of a general purpose data stream management system. The implementation techniques described in the paper include the representation of atomic application as sequences of operation in an XML based language and translation of XML specifications into the programs in an object-oriented programming language.

1 INTRODUCTION

The technological advances of small size and energy efficient electronic sensing devices allow for collecting and real time processing of long sequences of data items commonly known as *data streams*. A *data stream* is a theoretically unlimited and continuously expanding sequence of homogeneous data items (Babcock et al., 2002). Many of such sequences are obtained from the periodical measurements of parameters of physical processes like for instance the values of temperature, humidity, air pressure or even the series of radio signals received from the outer space. Processing and managing the high frequency data streams are beyond the performance capabilities of present commercial Database Management Systems (DBMS). It is commonly agreed that a new class of systems, commonly called as Data Stream Management Systems (DSMS) (Motwani et al., 2003), is needed to reach the performance levels needed by the data stream processing applications.

The typical application domains where it is hard to apply the general purpose DSMS include processing of data streams in the embedded systems and in the wireless sensor networks.

A methodology for the design and implementation of data stream processing applications in an object-oriented programming languages is still an open problem. The applications expressed in a high level query and data manipulation language must be translated into the programs in a lower level implementation language and later on these programs must be optimized as well. There is no well established and commonly accepted language suitable for programming and optimization of data stream processing applications at the implementation level.

An interesting question is how to formally express the functionality of a data stream processing application at the lower levels of abstraction and how to optimize an application ? As the data streams are theoretically unlimited sequences of data items, the processing of complete streams at any moment in time is practically impossible. To avoid this problem we assume that only subsets of data streams, also called as *windows*, are processed by an application. The *reactivity* principle requires the processing of all windows to be performed whenever the contents of at least one window have been changed. If only approximate results are expected then some of the data items from a window are processed and processing is performed every *n*-th modification.

This work is based on a formal model of data stream processing presented in (Getta and Vossough, 2004). If an application processes *n* data streams

$x_1, \ldots, x_n$ then we represent it as an n-ary operation $f(x_1, \ldots, x_n)$. To enforce the *reactivity* of an application we have to implement n operations, $f_1, \ldots, f_n$, each one on a data stream and $n-1$ windows. The operations are implemented as the expressions $e_1, \ldots, e_n$ built over the binary operations on the elements of data stream processed x_i and the windows on the streams $w_{x_1}, \ldots, w_{x_n}$.

Design and implementation of data stream processing applications in a way consistent with a formal model described above is suitable for the environments were data stream processing software has to be merged with software implemented in the general purpose programming languages.

The paper is organized in the following way. We start from a review of the previous works in an area of data stream processing systems. Section 3 presents the basic concepts of data stream model used in the paper. It is followed by a presentation of operational model of data stream processing. Section 4 and Section 5 that overviews the implementation aspects and the experiments conducted so far. Finally, Section 6 summarizes and concludes the paper.

2 PREVIOUS WORKS

Design and implementation of scalable and distributed data stream processing systems attracted a lot of attention in the last years. As many of the fundamental assumptions behind the traditional DBMS no longer hold for data stream processing systems (Babcock et al., 2002), implementation of the prototype systems presented a significant challenge.

STREAM system (Motwani et al., 2003) is a general purpose DSMS that supports a declarative query language and it is able to process many continuous queries on the data streams with high frequencies of input data. The system also supports the approximate query answering when processing the queries over the data streams with very high frequencies.

TelegraphCQ is a dataflow system for the processing of continuous queries over data streams (Avnur and Hellerstein, 2002). It uses an adaptive query engine, which is based on a concept of Eddy earlier invented for the adaptive query processing in the relational DBMS.

Aurora system (Abadi et al., 2003) is designed to process large number of asynchronous and push based data streams. Aurora builds continuous queries out of a small set of well defined operators that implement standard filtering, mappings, aggregate and windows join operations. Currently, the development of Aurora has been superseded by Borealis project, Borealis is a distributed data stream processing engine that inherits core functionality from Aurora and inter-node communication from Medusa system (Zdonik et al., 2003).

Gigascope is a data stream processing system for network applications including traffic analysis, intrusion detection, router configuration analysis, network monitoring, and performance monitoring and debugging (Cranor et al., 2003).

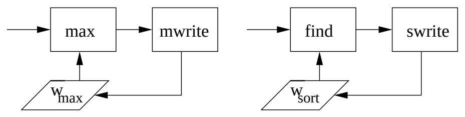

Figure 1: The compositions of elementary operations.

3 BASIC CONCEPTS

A *data stream* is a theoretically unlimited sequence of homogeneous and either elementary or composite data items. A type of the individual data items determines the special types of data streams, for instance, a *relational data stream* is a stream whose data items are the tuples of elementary values, *XML data stream* is a stream whose data items are the XML documents, and so on.

A system of elementary operations on data items is derived from a formal model of data stream processing and optimization proposed by (Getta and Vossough, 2004). An elementary operation always processes one input data stream, at most one data container, and outputs the data items to zero or more output data streams.

There are two types of data containers: *fixed size* container also called as *windows* on data streams and *variable size* containers used to keep the intermediate results of stream processing.

The complex operations on a data stream are implemented through the composition of elementary operations. For instance, finding the current largest value in a sequence of data items can be implemented as the composition of a *read* operation `max` and *write* operation `mwrite`, see Figure 1. In another example, an operation `find` compares an item taken from a stream with n largest and sorted items stored in a data container w_{sort} and finds all items smaller than the new one, see Figure 1. If at least one item is found a sequence of items that should be written to w_{sort} to keep it sorted is passed to an operation `swrite`,

In a general case implementation of n argument operation is performed by the decomposition of $f(w_1, \ldots, w_{i-1}, x_i, w_{i+1}, \ldots, w_n)$, into $n-1$ binary operations.

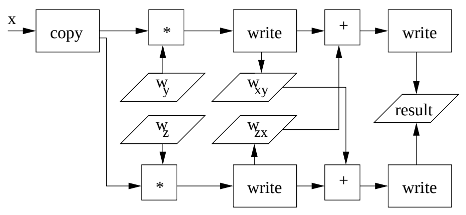

Figure 2: A data stream processing networks implementing $(x*w_y)+(x*w_z)$.

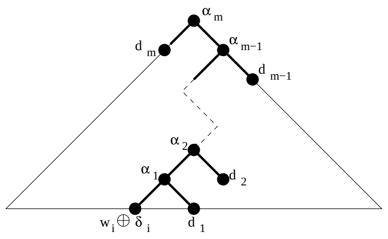

Figure 3: Implementation of $f_i(w_1,\ldots,w_i\oplus\delta_i,\ldots,w_n)$.

For example, an operation $f(x,w_y,w_z)$ that processes a data stream x, fixed size windows w_y, w_z on the streams y and z can be implemented as $f_2(f_1(x,w_y),w_z)$ and represented as a *path* $p : f_1(w_y), f_2(w_z), \varepsilon$. A data stream x is piped into a path p, $x \rightarrow p$ in order process the data items. The last symbol in a path identifies the next path to be used for the processing. A special symbol ε denotes an end of processing.

As a simple example consider an operation $f(x,w_x,w_y) = (x*w_y)+(x*w_z)$ that processes a data stream x against the fixed size windows w_y and w_z. The paths:
$p{:}copy()(1{:}p_1, 2{:}p_2)$
$p_1{:}*(w_y), write(w_{xy}), +(w_{zx}), write(result), \varepsilon$
$p_2{:}*(w_z), write(w_{zx}), +(w_{xy}), write(result), \varepsilon$
implementing $f(x,w_x,w_y)$ are visualized in Figure 2. A *module* is a set of paths encapsulated as a complex operation $m(p_1,\ldots,p_m,d_1,\ldots,d_n)1{:}\varepsilon,\ldots,p{:}\varepsilon$ where $p_1,\ldots,p_m$ are the path parameters, $d_1,\ldots,d_n$ are the data container parameters and $1{:}\varepsilon,\ldots,p{:}\varepsilon$ are the outputs.

Processing of many data streams needs the individual implementations of paths for each one of the streams involved in an application. A *data stream processing network* is a set of path expressions together with the data streams "piped" into the paths.

4 DESIGN OF APPLICATIONS

In an operational model of data stream processing an application acting on the streams $x_1,\ldots,x_n$ is represented as n-argument operation $f(x_1,\ldots,x_n)$. Due to the *reactivity* principle, an application should be able to recompute the operation after a new data item δ_i is appended to anyone of the streams. Therefore, an application programmer must provide the implementations of n operations $f_1(w_{x_1}\oplus\delta_1, w_{x_1},\ldots,w_{x_n}),\ldots, f_1(w_{x_1},\ldots,w_{x_n}\oplus\delta_n)$ where $w_{x_i}\oplus\delta_i$ denotes the contents of a window w_{x_i} after the insertion of a data item δ_i.

In order to speed up the evaluation of the operations, all computations on the *windows* that have not been changed since the previous evaluation are taken from the earlier recorded temporary results, also called as *materializations* $d_1,\ldots,d_m$, see Figure 3. Hence, the implementation of $f_i(w_1,\ldots,w_i\oplus\delta_i,\ldots,w_n)$ is performed through the transformation of n-argument operation into an expression $e_i(d_1,\ldots,d_m,w_i\oplus\delta_i)$ over the binary operations $\alpha_1,\ldots,\alpha_m$, *window* $w_i\oplus\delta_i$, and *materializations* $d_1,\ldots,d_m$,

A sequence of binary operations is transformed into a path $p_i : \alpha_1(d_1),\ldots,\alpha_m(d_m)$ where d_1 is a *window* on a data stream see Figure 3. A transformation of an expression e_i into a path is performed in the following way. We start from an operation $\alpha_1(\delta_i\oplus w_i, d_1)$ and we construct a path $p_i{:}\alpha_i(d_1)$. Next, we consider an operation $\alpha_2(\alpha_1(\delta_i\oplus w_i,d_1),d_2)$ and we extend a path p_i to get $p_i{:}\alpha_1(d_1)),\alpha_2(d_2)$. We repeat this process until an operation α_m at the root of expression e_i is processed. Finally, we append $write(w_{out})$ to a path p_i. We repeat, this process for all expressions e_i, $i=1,\ldots,n$. Next, if d_j is a materialization of the intermediate results then we insert an operation $write(d_j)$ into all paths expressions that contribute to the contents of d_j. At the end, we add an operation $write(w_i)$ at the beginning all paths p_i whose inputs are directly taken from the data streams.

5 IMPLEMENTATION OF APPLICATIONS

An implementation stage that follows an application design includes the preparation of formal specification and optimization of paths, generation of implementation code, and implementation of the operations. XML is chosen as a language for formal specification of paths.

XML document that describes a data stream processing application consists of `PORT`, `DATA-TYPE`, `WINDOWS`, and `PATH` elements. An element `PORT` includes information about the sources from where the data items are collected and it is described by the attributes `IP`, `TYPE`, and `TO-PATH`.

An element `DATA-TYPE` contains information about the structures of data items handled by an application. Its subelements and attributes are modeled in a way similar to C++ class structures.

An element `WINDOWS` contains information about the different types of windows used by an application.

An element `PATH` represents the paths a data stream processing application consists of. It is described by the attributes `NAME` and `TYPE` where `NAME` identifies a path and `TYPE` is a type of data item processed by a path. The subelements of `PATH` inlude the repetitions of the elements `OPERATION` and `OUTPUT`. An element `OPERATION` that represent the operations included in a path has its own sub-elements including `COMMENTS`,`GET`, `STORE`, and `BRANCH`.

C++ code generated from XML specification implements entire application except the elementary operations that have to be separately provided by the application programmers. Every data item handled by a data stream processing application has its type declared in the application. Data types are represented by C++ classes of objects and the variables being the instances of a particular data type are stored as either private or public variables.

All paths described in XML document are represented as sequences of operations are the segments within the function. A code generated from XML uses sockets for connecting and listening to different ports.

6 SUMMARY, AND FUTURE WORK

This work considers the design and implementation of data stream processing applications in the environments where the limited computational resources or specific requirements imposed on the applications make the utilization of complex DSMS not practical. In our approach an application processing n input data streams is represented as an n-ary operation. We show how to decompose such operations into the expressions built of binary operations, *materializations*, and input data items and later on we describe the transalation the expressions into the sets of data stream processing paths. In our model one data stream is directed for processing to one path and set of paths represents entire application. The paths are formally described in XML based language and implemented through the automatic translation into C++ code.

The following are the possible directions for future extensions of our approach to data stream processing. An interesting idea is to distribute the computations over many processing units. A closely related problem is the distribution of the processing in the sensor networks. Another problem is related to the simultaneous processing of more than one data stream. In such a case the synchronization of flows of data items along the processing paths needs to throughly be addressed.

REFERENCES

Abadi, D., Carney, D., Cetintemel, U., M.Cherniack, Convey, C., Erwin, C., Galvez, E., Hauton, M., Maskey, A., Rasin, A., A.Singer, Stonebraker, M., Tatbul, N., Xing, Y., Yan, R., and Zdonik, S. (2003). Aurora: A data stream management system. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 663–663.

Avnur, R. and Hellerstein, J. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 49–60.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In Popa, L., editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16. ACM Press.

Cranor, C., Johnson, T., Spatatschek, O., and Shkapenyuk, V. (2003). Gigascope: A stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 644–648.

Getta, J. R. and Vossough, E. (2004). Optimization of data stream processing. *SIGMOD Record*, 33(3):34–39.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., C.Olston, Rosenstein, J., and Varma, R. (2003). Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, pages 245–256.

Zdonik, S., Stonebraker, M., M.Cherniack, Cetintemel, U., Balazinska, M., and H.Balakrishnan (2003). The Aurora and Medusa projects. *Bulletin of the Technical Committee on Data Engineering*, pages 3–10.